CS1010 http://www.comp.nus.edu.sg/~cs1010/

Programming Methodology

UNIT 13

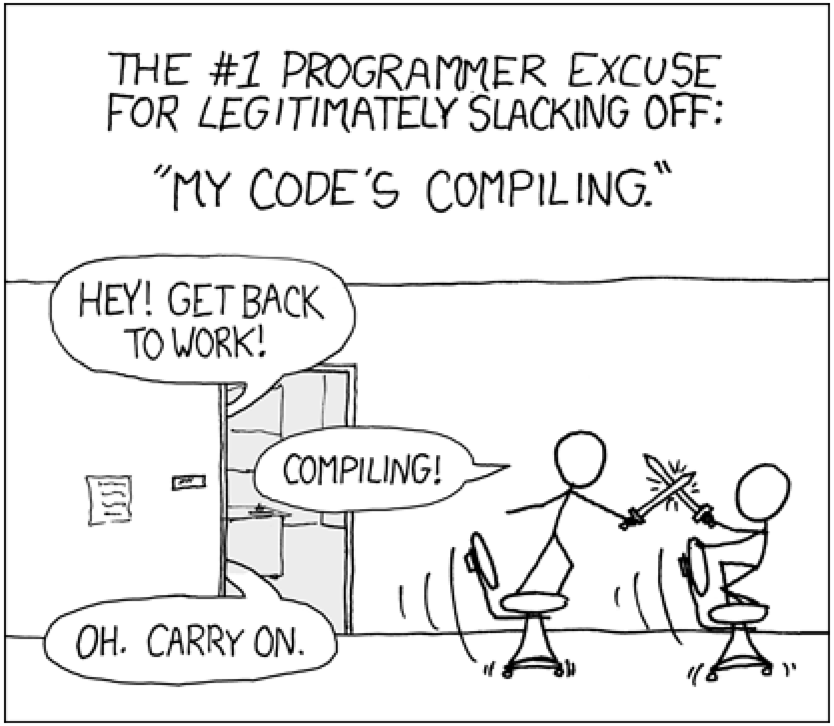
Separate Compilation

NUS National University of Singapore | School of Computing

Unit 13: Separate Compilation

Objective:

- Learn how to use separate compilation for program development

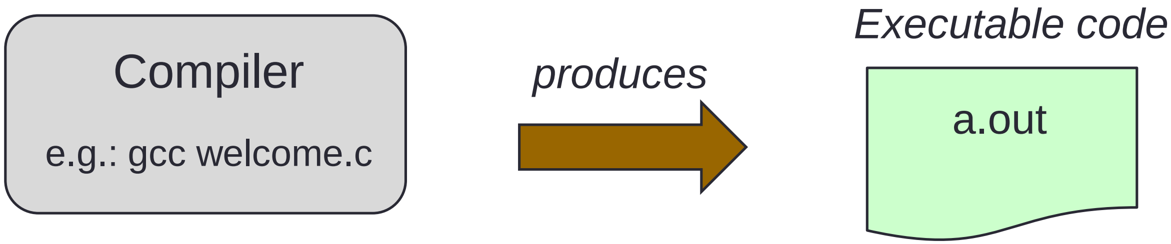


Unit 13: Separate Compilation

1. Introduction
2. Separate Compilation
3. Notes

1. Introduction (1/4)

- So far we have compiled our programs directly from the source into an executable:



- For the development of large programs with teams of programmers the following is practised
  - "Break" the program into multiple modules (files)
  - Compile the modules separately into object files (in C)
  - Link all object files into an executable file

1. Introduction (2/4)

- Header Files and Separate Compilation
  - Problem is broken into sub-problems and each sub-problem is tackled separately – divide-and-conquer
  - Such a process is called modularization
  - The modules are possibly implemented by different programmers, hence the need for well-defined interfaces
  - The function prototype constitutes the interface (header file). The function body (implementation) is hidden – abstraction
  - Good documentation (example: comment to describe what the method does) aids in understanding

1. Introduction (3/4)

- Example of documentation
  - The function header is given
  - A description of what the function does is given
  - How the function is implemented is not shown

```
double pow(double x, double y);
// Returns the result of raising
// x to the power of y.
```

C library function - pow()

Previous Page Next Page

Description

The C library function **double pow(double x, double y)** returns **x** raised to the power of **y** i.e. x^y.

Declaration

Following is the declaration for pow() function.

```
double pow(double x, double y)
```

Parameters

- **x** -- This is the floating point base value.
- **y** -- This is the floating point power value.

Return Value

This function returns the result of raising x to the power y.

Example

The following example shows the usage of pow() function.

Try it

```
#include <stdio.h>
#include <math.h>

int main ()
{
   printf("Value 8.0 ^ 3 = %lf\n", pow(8.0, 3));

   printf("Value 3.05 ^ 1.98 = %lf", pow(3.05, 1.98));

   return(0);
}
```

1. Introduction (4/4)

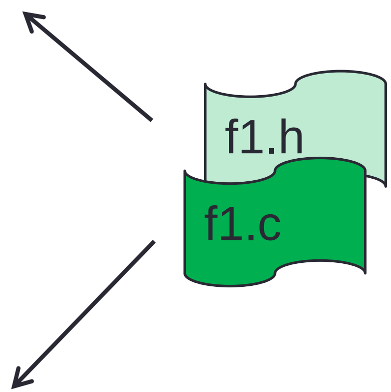
- Reason for Modular Programming
  - Divide problems into manageable parts
  - Reduce compilation time
    - Unchanges modules do not eed to be re-compiled
  - Facilitate debugging
    - The modules can be debugged separately
    - Small test programs can be written to test the functions in a module
  - Build libraries of useful functions
    - Faster development
    - Do not need to know how some functionality is implemented, e.g., image processing routines
    - Example: OpenCV – a computer vision library.

2. Separate Compilation (1/2)

- From http://encyclopedia2.thefreedictionary.com/
- Separate Compilation:
  - A feature of most modern programming languages that allows each program module to be compiled on its own to produce an object file which the linker can later combine with other object files and libraries to produce the final executable file.
- Advantages
  - Separate compilation avoids processing all the source code every time the program is built, thus saving development time. The object files are designed to require minimal processing at link time. The can also be collected together into libraries and distributed commercially without giving away source code (through they can be disassembled).
- Examples of output of separate compilation:
  - C object files (.o files) and Java .class files.

2. Separate Compilation (2/2)

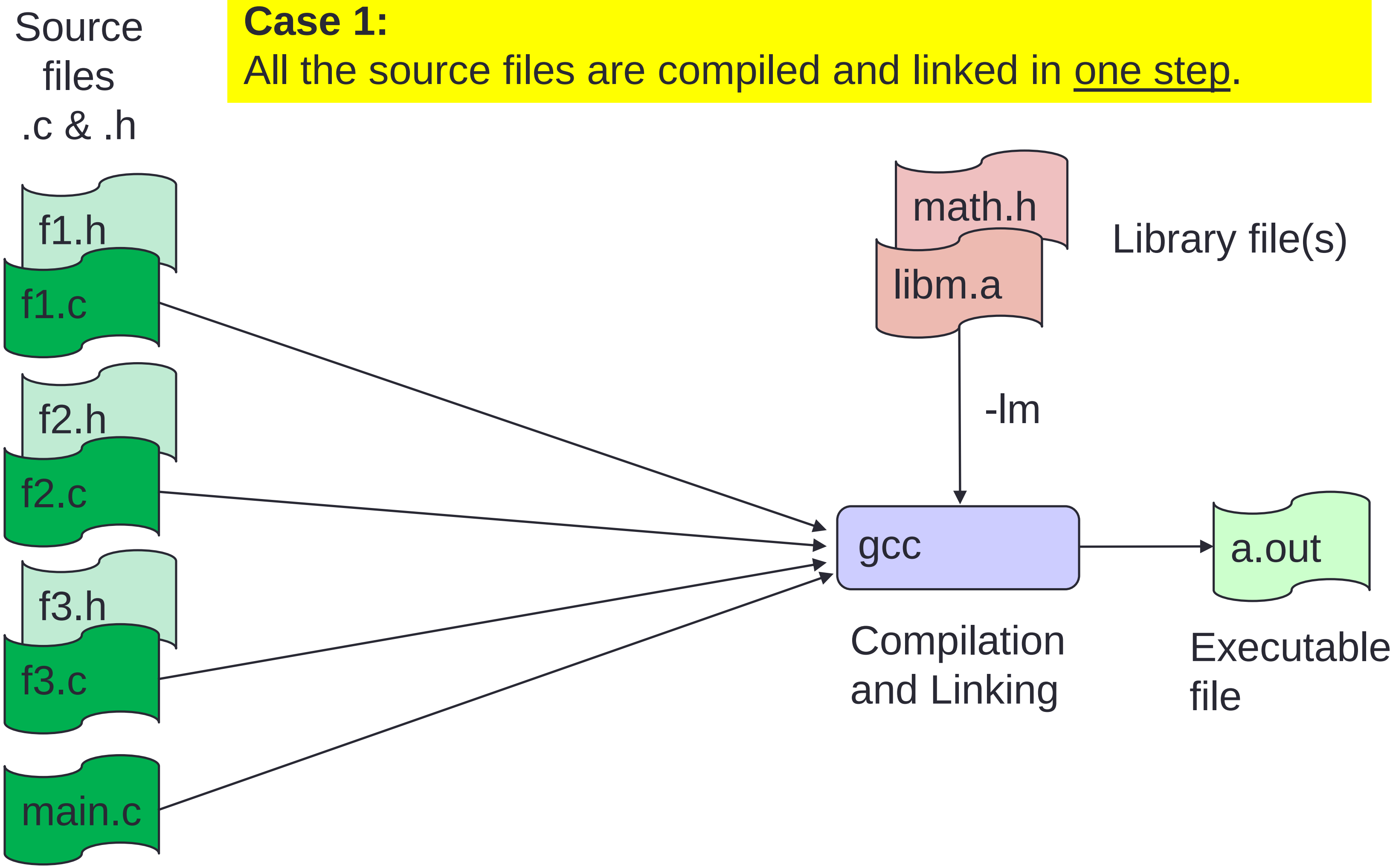
- In most cases, a module contains functions that are related, e.g., math functions.
- A module consists of
  - A header file (e.g. f1.h) which contains:
    - Constant definitions, e.g.:
      - `#define   MAX   100`
    - Function prototypes, e.g.:
      - `double mean(double, double);`



  - A source file (e.g. f1.c) which contains:
    - The functions that implement the function prototypes in the header file (e.g., the code for the function `mean(…)`).
    - Other functions, variables, and constants that are only used within the module (i.e., they are module-local).

2.1 Separate Compilation: Case 1

Case 1:
All the source files are compiled and linked in <u>one step</u>.



2.1 Case 1 Demo

- Let's re-visit the Freezer version 2 program in Unit 4 Exercise 6. We will create a module that contains a function to calculate the freezer temperature:
  - Module header file:

Unit13_FreezerTemp.h

```
// Compute new temperature in freezer
float calc_temperature(float);
```

  - Module source file:

Unit13_FreezerTemp.c

```
#include <math.h>

// Compute new temperature in freezer
float calc_temperature(float hr) {
   return ((4.0 * pow(hr, 10.0))/(pow(hr, 9.0) + 2.0)) - 20.0;
}
```

2.1 Case 1 Demo: Main Module

Unit13_FreezerMain.c

```
#include <stdio.h>
#include "Unit13_FreezerTemp.h"

int main(void) {
    int hours, minutes;
    float hours_float;  // Convert hours and minutes into hours_float
    float temperature;  // Temperature in freezer

    // Get the hours and minutes
    printf("Enter hours and minutes since power failure: ");
    scanf("%d %d", &hours, &minutes);

    // Convert hours and minutes into hours_float
    hours_float = hours + minutes/60.0;

    // Compute new temperature in freezer
    temperature = calc_temperature(hours_float);

    // Print new temperature
    printf("Temperature in freezer = %.2f\n", temperature);

    return 0;
}
```

Include the header file (Note "..." instead of <…>).
Header file should be in the same directory as this program.

Now we can write a program which uses the new external function

2.1 Case 1 Demo: Compile and Link

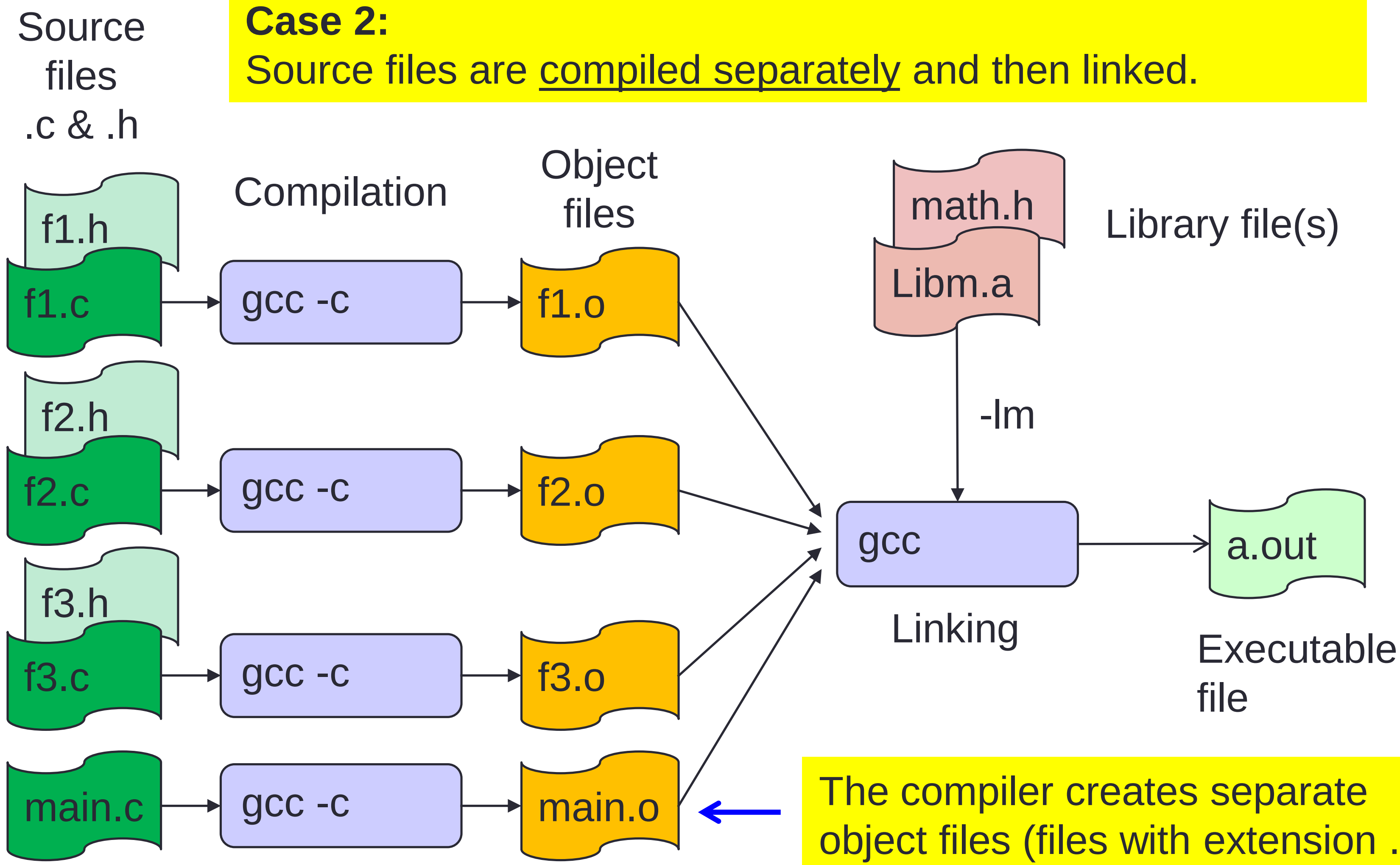
- How do we run Unit13_FreezerMain.c, since it doesn't contain the function definition of calc_temperature()?
- Need to compile and link the programs

```
$ gcc Unit13_FreezerMain.c Unit13_FreezerTemp.c -lm
```

- Here, the compiler creates temporary object files (which are immediately removed after linking) and directly creates a.out
- Hence, you don't get the chance to see the object files (files with extension .o)
- (Note: The option –Wall is omitted above due to space constraint. Please add the option yourself.)

2.2 Separate Compilation: Case 2

Case 2:
Source files are compiled separately and then linked.



The compiler creates separate object files (files with extension .o)

2.2 Case 2 Demo: Compile and Link

- For our Freezer program:

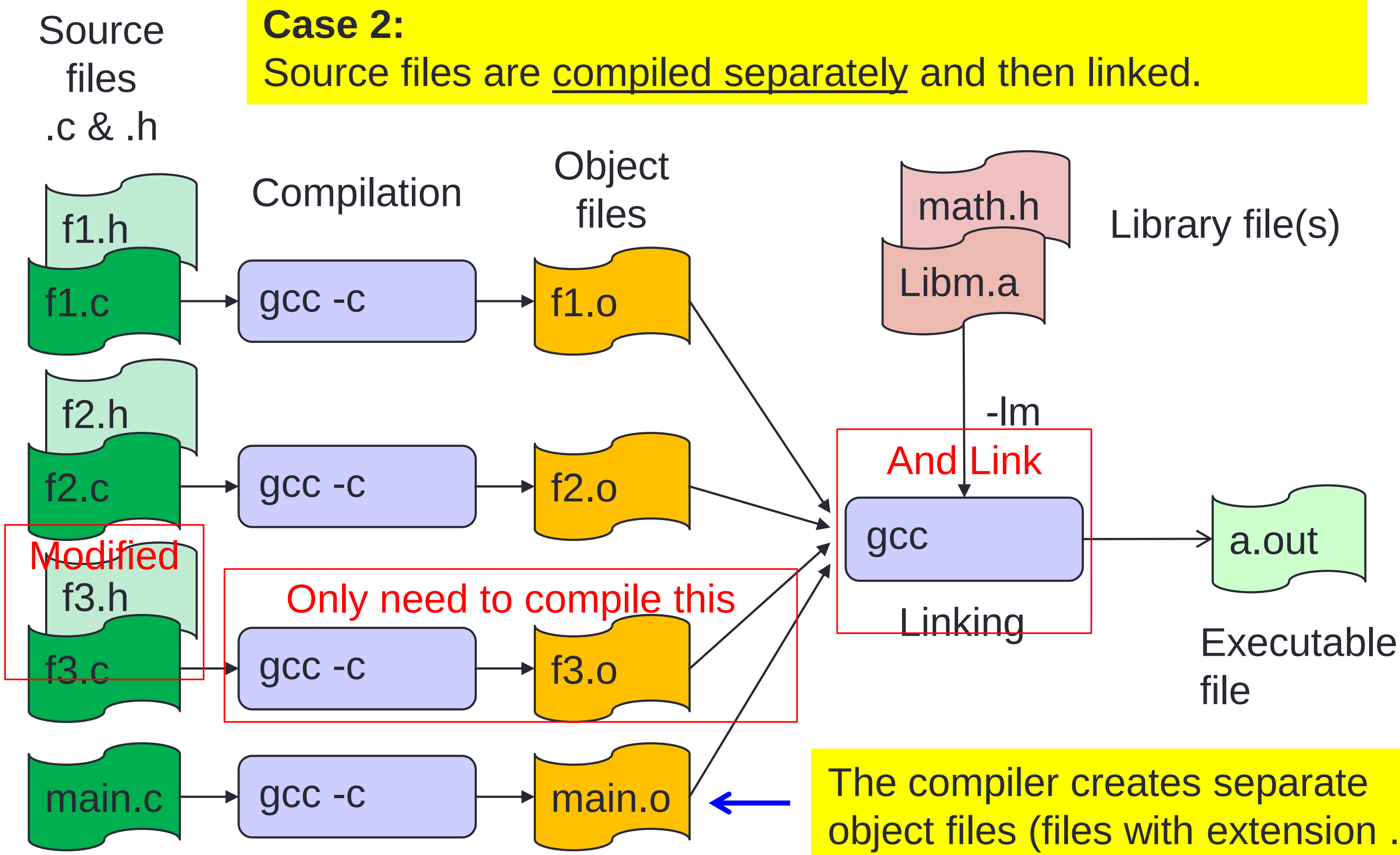
```
$ gcc –c Unit13_FreezerMain.c
$ gcc –c Unit13_FreezerTemp.c
$ gcc Unit13_FreezerMain.o Unit13_FreezerTemp.o  -lm
```

- Here, we first create the Unit13_FreezerMain.o and Unit13_FreezerTemp.o object files, using the –c option in gcc.
- Then, we link both object files into the a.out executable
- (Note: The option –Wall is omitted above due to space constraint. Please add the option yourself.)

```
basic:
arg.c                cmdlookup.cpp        farray.hpp           license.cpp          nurbsdata.cpp        rarray.h             rgitypes.h           versions.h
basic.c              cmdlookup.h          files.c              license.h            nurbsdata.h          rarray.hpp           rgivector.cpp        vertexarray.cpp
basic.dsp            command.cpp          flexlm.cpp           list.h               orindex.h            rectsel.cpp          rgivector.h          vertexarray.h
basic.h              command.h            flexlm.h             list.hpp             orindex.hpp          rectsel.h            rgivector.hpp        vltdata.cpp
basic.plg            comment.cpp          freearray.h          lm_attr.h            ortri.h              rgicstring.cpp       spectrum.cpp         vltdata.h
binio.cpp            comment.h            freearray.hpp        lm_code.h            ortri.hpp            rgicstring.h         spectrum.h           wfshortestpather.cpp
binio.h              console.cpp          genmatrix.cpp        lmclient.h           perftimer.cpp        rgicstring.hpp       stackbv.cpp          wfshortestpather.h
binio.hpp            console.h            genmatrix.h          lmpolicy.h           perftimer.h          rgimatrix.cpp        stackbv.h            win_basic.h
bitvector.cpp        convert.cpp          getarg.c             logfile.cpp          points.cpp           rgimatrix.h          stringtable.cpp      wireframe.cpp
bitvector.h          convert.h            history.cpp          logfile.h            points.h             rgimatrix.hpp        stringtable.h        wireframe.h
bitvector.hpp        convert.hpp          history.h            malloc.c             points.hpp           rgimessage.cpp       time.c               wireframe.hpp
build.h              data.cpp             iit.c                map.h                pqueue.h             rgimessage.h         tokenize.c           xdr.c
callbacklist.cpp     data.h               index.h              map.hpp              pqueue.hpp           rgimessagestack.cpp  tritype.h            xdr.h
callbacklist.h       dumpable.cpp         isort.c              math2.c              prime.c              rgimessagestack.h    uf.c
callbackobject.cpp   dumpable.h           iterstack.h          miscmath.cpp         qsort.c              rgistring.cpp        unix_basic.h
callbackobject.h     dumpable.hpp         iterstack.hpp        miscmath.h           queue.h              rgistring.h          util.h
cb_doprnt.c          facepoint.h          kdtree.cpp           multitree.h          queue.hpp            rgitranslator.cpp    vectmat.cpp
cb.c                 farray.h             kdtree.h             multitree.hpp        raindrop.c           rgitranslator.h      vectmat.h

CompDB:
compDB.cpp  CompDB.dsp  compDB.h    CompDB.plg

delone:
boundary.cpp   dcbuilder.cpp  dcofaces.cpp   dcomp.cpp      dcompiter.cpp  delone.dsp     faces.cpp      ksimpsize.h    simpsize.h
boundary.h     dcbuilder.h    dcofaces.h     dcomp.h        dcompiter.h    delone.plg     ksimpsize.cpp  simpsize.cpp

geometry:
animate.cpp        comp.cpp           edgeset.h          ihandler.cpp       modtrinfo.hpp      segmenttree.cpp    simplex.h          trist.cpp          vertarray.h
animate.h          comp.h             fliphandler.cpp    ihandler.h         orienter.cpp       segmenttree.h      simplexset.cpp     trist.h            vertex.cpp
boxes.cpp          computil.cpp       fliphandler.h      ksimplex.cpp       orienter.h         shortestpather.cpp simplexset.h       trist.hpp          vertex.h
boxes.h            computil.h         geometry.dsp       ksimplex.h         ortribv.cpp        shortestpather.h   testint.cpp        tristconnector.cpp vertset.cpp
bvtag.cpp          edgecycleset.cpp   geometry.plg       locate.cpp         ortribv.h          simph.cpp          testint.h          tristconnector.h   vertset.h
bvtag.h            edgecycleset.h     geomutil.cpp       locate.h           packedihandler.cpp simph.h            tolerancer.cpp     tristmodifier.h
cofaces.h          edgeset.cpp        geomutil.h         modtrinfo.h        packedihandler.h   simplex.cpp        tolerancer.h       vertarray.cpp

li:
base.h      det.c       li.dsp      li.hpp      lia.c       liaux.c     lidet.cpp   liminor.cpp lipoints.cpp listack.cpp pool.c
chars.c     li.cpp      li.h        li.plg      lia.h       liaux.c.old lidet.h     liminor.h   lipoints.h   listack.h   stack.c

Skin:
a.h                  ChildFrm.cpp         FormCommandView.cpp  MainFrm.h            resource.h           Skin.dsw             Skin.plg             SkinView.cpp
AlphaDlg.cpp         ChildFrm.h           FormCommandView.h    ReadMe.txt           Skin.aps             Skin.h               Skin.rc              SkinView.h
AlphaDlg.h           dump.stl             InputCQ.cpp          RenderView.cpp       Skin.clw             skin.log             Skin.reg             StdAfx.cpp
beforeRefinement.sav FileOpenOption.cpp   InputCQ.h            RenderView.h         Skin.cpp             Skin.ncb             SkinDoc.cpp          StdAfx.h
beforeRefinement.stl FileOpenOption.h     MainFrm.cpp          res                  Skin.dsp             Skin.opt             SkinDoc.h
```

2.2 Separate Compilation: Case 2

Case 2:
Source files are <u>compiled separately</u> and then linked.



The compiler creates separate object files (files with extension .o)

2.2 Case 2 Demo: Compile and Link

- For our Freezer program:

```
$ gcc -c Unit13_FreezerMain.c
$ gcc -c Unit13_FreezerTemp.c
$ gcc Unit13_FreezerMain.o Unit13_FreezerTemp.o  -lm
```

- Let's say if you only modified Unit13_FreezerTemp.c but NOT Unit13_FreezerMain.c, you can skip the first compilation

```
$ gcc -c Unit13_FreezerMain.c
$ gcc -c Unit13_FreezerTemp.c
$ gcc Unit13_FreezerMain.o Unit13_FreezerTemp.o  -lm
```

- Speed of a lot if you have tons of files

3. Notes (1/2)

- Difference between
  - #include < … > and #include " … "
  - Use " … " to include your own header files and < … > to include system header files. The compiler uses different directory paths to find < … > files.
- Inclusion of header files
  - Include *.h files only in *.c files, otherwise duplicate inclusions may happen and later may create problems:
    - Example: Unit13_FreezerTemp.h includes <math.h>
      Unit13_FreezerMain.c includes <math.h> and "Unit13_FreezerTemp.h"

      Therefore, Unit13_FreezerMain.c includes <math.h> twice.

3. Notes (2/2)

- ‘Undefined symbol’ error
  - ld: fatal: Symbol referencing errors.
  - The linker was not able to find a certain function, etc., and could not create a complete executable file.
    - Note: A library can have missing functions → it is not a complete executable.
  - Usually this means you forgot to link with a certain library or object file. This also happens if you mistyped a function name.

Summary

- In this unit, you have learned about
  - How to split a program into separate modules, each module containing some functions
  - How to separately compile these modules
  - How to link the object files of the modules to obtain the single executable file

End of File